Electronic version of an article published as [Journal of the Operational Research Society, 2008, vol. 59, p. 812-822]
[DOI: http://dx.doi.org/10.1057/palgrave.jors.2602413]

A Variable Neighbourhood Search Algorithm for the Constrained Task Allocation Problem

Amaia Lusa[*1]*; Chris N. Potts*[2]

[1]*Research Institute (IOC) / Engineering School (ETSEIB)*

Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

amaia.lusa@upc.edu

[2]*School of Mathematics, University of Southampton,, Southampton, UK*

C.N.Potts@maths.soton.ac.uk

ABSTRACT

A Variable Neighbourhood Search algorithm that employs new neighbourhoods is proposed for solving a task allocation problem whose main characteristics are: (*i*) each task requires a certain amount of resources and each processor has a capacity constraint which limits the total resource of the tasks that are assigned to it; (*ii*) the cost of solution includes fixed costs when using processors, task assignment costs, and communication costs between tasks assigned to different processors. A computational study shows that the algorithm performs well in terms of time and solution quality relative to other local search procedures that have been proposed.

Keywords: task allocation problem, variable neighbourhood search, local search.

Introduction

The task allocation problem (TAP) consists of assigning a set of tasks to a set of processors (or machines) so that the overall cost is minimized. This cost may include a fixed cost for using a processor, a task assignment cost that may depend on the task and processor, and a communication cost between tasks that are assigned to different processors. The problem can be constrained (CTAP) or unconstrained (UTAP) depending on whether or not processors have a limited capacity.

* Corresponding author: Amaia Lusa, Research Institute IOC, Av. Diagonal 647 (edif. ETSEIB), p.11, 08028 Barcelona, Spain; Tlf. + 34 93 401 17 05; Fax. + 34 93 401 66 05 ; e-mail: amaia.lusa@upc.edu

Specifically, for the CTAP, each task has an associated resource requirement, and each processor has a capacity constraint which limits the total resource of the tasks that are assigned to it.

The problem arises in distributed computing systems (Stone, 1977) where a number of tasks (programs, editing files, managing data, etc.) are to be assigned to a set of processors (computers, disks, etc.) to guarantee that all tasks are executed within a certain cycle time. The aim is to minimise the cost of the processors and the inter-processor data communication bandwidth installed. The problem has also many industrial applications. For example, according to Hadj-Alouane et al. (1999), Rao, in 1992, introduces a specific constrained task allocation problem in the automobile manufacturing industry: in the modern automobile, many tasks such as integrated chassis and active suspension monitoring, fuel injection monitoring, etc., are performed by a subsystem consisting of micro-computers linked by high-speed and/or low speed communication lines. The cost of the subsystem is the sum of costs of the micro-computers (or processors), and the installation costs of the data links that provide inter-processor communication bandwidth. Each task deals with the processing of data coming from sensors, actuators, signal processors, digital filters, etc., and has a throughput requirement in KOP (thousand operations per second). Several types of processors are available and, for each, is known its purchase cost and its throughput capacity in terms of the KOP it can handle. The tasks are interdependent; a task may need data from another task to be completed. Hence, if two tasks are assigned to different processors, they may need a communication link with a certain capacity. The communication load between two tasks is independent of the processors to which they are assigned.

Since its introduction by Stone (1977), many authors have tackled different versions of the problem by applying exact algorithms, heuristic procedures, and meta-heuristics. However, only a few studies have dealt with the constrained version (Chen and Lin, 2000; Hadj-Alouane et al., 1999; Hamam and Hindi, 2000; Ernst et al., 2006), and, due to the complexity of the problem, none of them are capable of solving some real-world applications optimally. To date, the best among known approaches for the CTAP is the hybrid method developed by Chen and Lin (2000), who combine tabu search and a noising method in their algorithm.

Variable neighbourhood search (VNS) is a relatively recent meta-heuristic for obtaining near-optimal solutions to combinatorial optimization problems (Mladenovic and Hansen, 1997) whose main feature is the systematic change of neighbourhood within a local search procedure. Different versions of VNS have been successfully applied to a variety of problems such as *bin-packing*, the *p-median problem*,

the *quadratic assignment problem*, the *travelling salesman problem* and the *vehicle routing problem*. We refer to Hansen and Mladenovic (2001) for a review of the technique and applications.

In this paper we propose an algorithm based on a VNS scheme to solve the CTAP. Through the use of five different neighbourhoods, our algorithm has the capability to navigate the solution space more effectively than previously proposed neighbourhood search methods. The results of a computational study show that our procedure outperforms the hybrid method developed by Chen and Lin (2000).

This paper is organised as follows. The next section introduces the CTAP, and presents the local search methods that have been proposed previously. We then describe the VNS approach in general, and develop our VNS algorithm for the CTAP. The penultimate section describes our computational experiments and reports the main results. Finally, we provide some conclusions in last section.

The constrained task allocation problem

The CTAP consists of assigning n tasks to m processors, subject to processor capacity constraints. The goal is to minimise the total cost, which comprises costs of assigning tasks to processors, fixed costs for using the processors, and communication costs for tasks assigned to different processors. The following quantities comprise an instance of the CTAP:

n	number of tasks
m	number of processors
a_i	resource requirement of task i (i=1,…,n)
b_k	capacity of processor k (k=1,…,m)
d_{ik}	cost of assigning task i to processor k (i=1,...,n; k=1,...,m)
s_k	fixed cost of using processor k (k=1,...,m)
c_{ij}	communication cost if tasks i and j are assigned to different processors (i=1,...,n; j=1,…,n); this cost is independent of the processors involved.

To specify the problem more precisely, we present a zero-one programming formulation, which uses the following variables:

$x_{ik} \in \{0,1\}$ indicates whether task i is assigned to processor k (i=1,...,n; k=1,...,m)

$y_k \in \{0,1\}$ indicates whether any task is assigned to processor k (k=1,...,m)

The formulation, which has a quadratic objective function, is as follows:

$$[MIN]z = \sum_{i=1}^{n-1}\sum_{j=1}^{n} c_{ij}\left(1 - \sum_{k=1}^{m} x_{ik}x_{jk}\right) + \sum_{i=1}^{n}\sum_{k=1}^{m} d_{ik}x_{ik} + \sum_{k=1}^{m} s_k y_k \quad (1)$$

$$\sum_{k=1}^{m} x_{ik} = 1 \qquad i = 1,...,n \quad (2)$$

$$x_{ik} \le y_k \qquad i = 1,...,n;\ k = 1,...,m \quad (3)$$

$$\sum_{i=1}^{n} a_i x_{ik} \le b_k y_k \qquad k = 1,...,m \quad (4)$$

Equation (1) is the allocation cost to be minimised (communication, assignment and fixed costs); (2) is the constraint that each task is to be assigned to one and only one processor; (3) ensures that the binary variable y_k takes value 1 if any task is assigned to processor k; and (4) expresses the processor capacity constraints.

When the number of processors is equal to 2, the problem can be transformed into a minimum cost cut problem (Stone, 1977) and optimally solved using network flow techniques. However, the problem has been shown to be NP-hard when the number of processors is equal or greater than 3 (Rao et al., 1979).

Since the work of Stone (1977), great progress has been made both in computer power and computational technology. Ernst et al. (2006) explore the potential of mathematical programming approaches and develop different formulations for the UTAP and CTAP. Nevertheless, results for the CTAP indicate that these approaches cannot be considered satisfactory for practical instances. Hence, some type of heuristic or meta-heuristic approach seems appropriate for tackling the CTAP and finding near-optimal solutions.

Various studies propose local search procedures to solve different versions of the constrained problem. Hadj-Alouane et al. (1999) develop a hybrid of Lagrangian relaxation and genetic algorithm that is subsequently shown not to be very efficient when compared to other procedures (Chen and Lin, 2000). Hamam and Hindi (2000) propose a simulated annealing algorithm. Their computational experience is very limited and there are no results to assess the effectiveness of their algorithm in terms of quality solution. Finally, Chen and Lin (2000) propose a hybrid approach which combines a tabu search and a noising method. Essentially, there are three major steps in their approach. First, a relaxed initial solution is created, which assigns all tasks to the cheapest processor (lower fixed cost). Second, a local search is performed which first uses tabu search, and then tries to improve on the best solution found by the repeated iterative process of adding noise to the communication costs, applying descent to find a local optimum with the perturbed communication costs, and then applying descent to find a local optimum with the original communication costs. In the final step, a processor substitution technique is applied to improve the solutions. Each component of the local search (tabu search and noising) is run in two phases: the first uses as a neighbourhood those solutions in which a task is reallocated to another processor; and the second uses as a neighbourhood those solutions in which two tasks, allocated to different processors, are exchanged. The results of computational experiments with a set of randomly generated instances lead them to conclude that their algorithm is superior to a random method, to tabu search, to the noising method and to the genetic algorithm of Hadj-Alouane et al. (1999) both in terms of solution quality and computation time. All the aforementioned algorithms allow non-feasible solutions. Constraint violations are handled by adding appropriate penalties and the algorithms obtain feasible solutions in practice, although feasibility is not guaranteed.

Our major concern about previous local search procedures is the neighbourhoods they consider. These algorithms consider the processor in which each task is allocated in the current solution and attempt the following moves: (1) reallocate a task to another processor; and (2) exchange two tasks assigned to different processors. Although, theoretically speaking, it is possible to achieve any solution by forming a sequence of these moves, some of the individual moves may be too bad to be performed and hence some solutions may remain unexplored. For example, assigning only one task to an empty processor is often a very bad move (due to the fixed costs), but a good move could consist of allocating a group of tasks with high communication costs to an empty processor. Thus, the algorithms often produce local optima after a short execution time, whereas this problem could be partially avoided through the use of other types of moves (reallocating a group of tasks, for example).

We add the three following types of neighbourhood to the ones traditionally used (reallocating a task and exchanging two tasks) when solving TAP, allowing us to explore interesting regions of the

solution space: (1) reallocating a cluster of tasks from one processor to another; (2) reallocating a cluster of tasks from different processors to another processor; and (3) emptying a processor by reallocating its assigned tasks to other processors. The results obtained by including these neighbourhood structures in a VNS algorithm are very satisfactory.

The variable neighbourhood search algorithm

One of the most successful versions of the VNS is the General Variable Neighbourhood Search, GVNS (Hansen et al., 2003), which is outlined in Figure 1. The termination condition can be either a maximum CPU time or a maximum number of iterations between two consecutive improvements. One of the steps of GVNS is a descent local search using different neighbourhoods, VND, which is outlined in Figure 2. VND terminates when no improvement is possible, thereby giving a solution that is a local optimum in all of the neighbourhoods that are used.

We make use of the following notation: x is the initial solution; $f(x)$ is the cost of solution x; u_{max} is the number of neighbourhood structures applied; and $N_u(x)$ is the neighbourhood of type u of solution x (u=1,…,u_{max}). To improve efficiency, f(x) is updated from its previous value in each step (not re-evaluated).

Insert Figure 1. General Variable Neighbourhood Search Algorithm, GVNS

Insert Figure 2. Variable Neighbourhood Descent Algorithm, VND

Neighbourhoods

Five neighbourhood structures have been used with the aim of allowing the algorithm to explore different regions of the solution space. None of the following moves allow infeasible solutions. Hence, it is guaranteed that the algorithm always gives a feasible solution (in contrast to Hadj-Alouane et al., 1999, and Chen and Lin, 2000 who allow exploration of infeasible solutions). Neighbourhoods N_1 and N_2 are well known, while N_3, N_4 and N_5 are new. In the description below, x denotes the current solution, i and j are tasks, and k and l are processors.

$N_1(x)$ reallocate a task i from processor k to processor l.

$N_2(x)$ exchange two tasks (task i from processor k to processor l and task j from processor l to processor k).

$N_3(x)$ reallocate a cluster of tasks from processor k to processor l.

$N_4(x)$ reallocate a cluster of tasks from different processors to processor l.

$N_5(x)$ empty processor k.

Communication and assignment costs are considered when determining the cluster of tasks to be reallocated. A full description of the moves considered under the three new types of neighbourhood N_3, N_4 and N_5 proposed in our GVNS are described below.

We make use of the following notation:

x_u a solution belonging to $N_u(x)$ (u=1,…,5)

P_k a set of tasks currently assigned to processor k (k=1,…,m)

$b_k^{'}$ the remaining capacity of processor k (k=1,…,m)

T_{kl} a cluster (set of tasks) currently assigned to processor k that can be assigned to processor l (k=1,…,m; l=1,…,m | $l \neq k$)

T_l a cluster (set of tasks) that can be assigned to processor l (l=1,…,m)

In Figure 3, an algorithm to find a neighbour $x_3 \in N_3(x)$ is presented. In the computation of C_j, the costs added correspond to "attracting" task j to processor l, while the costs subtracted correspond to "attracting" task j to the processor where it is currently assigned. The idea of setting an initial task s to begin a cluster is to allow a set of tasks with high communication costs to be reallocated together. Without an initial task, the reallocation process would be driven by the costs of assigning tasks to processors. Neighbourhood $N_3(x)$ is obtained by selecting all pairs of processors k and l and, for each pair, choosing at random a different task s to initiate a cluster. To avoid too many cluster repetitions, each task is selected with probability 0.7 to initiate a cluster. Furthermore, the parameter α is chosen randomly by the clustering algorithm with the aim of creating some diversification.

To find $x_4 \in N_4(x)$ and $x_5 \in N_5(x)$, similar ideas are employed to those for finding x_3. Details are given in Figures 4 and 5, respectively.

Insert Figure 3. Procedure to find x_3

Insert Figure 4. Procedure to find x_4

Insert Figure 5. Procedure to find x_5

Size of neighbourhoods

Next we provide the size, denoted by Size_u, of each neighbourhood N_u used in the GVNS algorithm, together with the time complexity of searching the neighbourhood.

N_1 There are n tasks and each can be moved to $m-1$ machines. Therefore, $\text{Size}_1 = O(mn)$. Since each neighbour is evaluated in $O(n)$ time, the time complexity to search this neighbourhood is $O(mn^2)$.

N_2 There are $O(n^2)$ pairs of tasks for selection. Therefore, $\text{Size}_2 = O(n^2)$. Since each neighbour is evaluated in $O(n)$ time, the time complexity to search this neighbourhood is $O(n^3)$.

N_3 There are n tasks, each of which can start a cluster on the machine to which it is allocated, and there are $m-1$ possible machines to which this cluster can be moved. Once the set T_{kl} is initialized, the remaining tasks for inclusion in T_{kl} are determined by our procedure and the neighbour evaluated in $O(n^2)$ time. Therefore, $\text{Size}_3 = O(mn)$, and the time complexity to search this neighbourhood is $O(mn^3)$.

N_4 There are m choices for the processor l, and $O(n)$ ways of choosing a task to start the corresponding cluster. As for N_3, the tasks to be reallocated are determined once T_l is initialized, and this requires $O(n^2)$ time including neighbour evaluation. Therefore, $\text{Size}_4 = O(mn)$, and the time complexity to search this neighbourhood is $O(mn^3)$.

N_5 There are m choices for the processor k, and the tasks to be reallocated are then determined and the neighbour evaluated in $O(mn^2)$ time. Therefore, $\text{Size}_5 = O(m)$, and the time complexity to search this neighbourhood is $O(m^2n^2)$.

Some implementation details affect the actual numbers of neighbours explored. The procedures to find x_3, x_4 and x_5 can give different neighbours if more than one value is used for α. To reduce computing time only one value was used in the experiments. Also, as indicated above, for N_3 and N_4, each potential task for starting a cluster is selected with probability 0.7.

Initial solution

The same basic ideas included in clustering procedures for our new neighbourhoods are also used to obtain a starting solution for GVNS. Although random solutions give good results, preliminary experiments show that on average the procedure outlined in Figure 6 is better.

Insert Figure 6. Algorithm to find initial solution

Computational experiments

The objective of our computational experiments is to evaluate the efficiency and effectiveness of the proposed GVNS algorithm. Specifically, we aim to assess whether the algorithm gives good solutions in a reasonable computation time even for large instances, and to compare the quality of the solutions obtained with the best known procedure, which is the hybrid method developed by Chen and Lin (2000).

Ernst et al. (2006) and Hadj-Alouane et al. (1999) report results for 8 real-world instances from an automobile microcomputer system and a Hughes air-defence system. Chen and Lin (2000) describe a way of constructing problem instances by randomly generating the data. The assignment costs d_{ik} are not considered in any of these studies. We coded the hybrid method (HYBRID) of Chen and Lin (2000), and included assignment costs d_{ik} by adding them to the objective function used by HYBRID, and ran three experiments as follows.

1. Apply GVNS and HYBRID to the 8 real-world instances provided by Hadj-Alouane, Bean and Murty, and compare these results with the ones obtained by Hadj-Alouane et al. (1999) and Ernst et al. (2006).

2. Apply GVNS and HYBRID to a new set of 108 randomly generated instances, without considering assignment costs (so the HYBRID is exactly the algorithm described by Chen and Lin (2000)).

3. Apply GVNS and the HYBRID to a new set of 54 randomly generated instances, including assignment costs.

Each algorithm is run 50 times and, to get a fair comparison, the maximum solving time of HYBRID is recorded. The two following versions of GVNS with different termination criteria are considered:

(a) GVNS1: Use as a termination condition a maximum number of iterations between two consecutive improvements, which we set to n, and a maximum computation time, which we set to be the maximum HYBRID solving time, and

(b) GVNS2: Use as a termination condition a maximum number of iterations between two consecutive improvements, which we set to n, and a maximum computation time of 50 seconds.

Note that in both GVNS1 and GVNS2, the computation time is checked after each iteration, and therefore the actual time may slightly exceed the time limit set for termination. For HYBRID, approximately $n/2$ iterations are performed, where one iteration comprises adding noise to the communication costs, applying descent to obtain a local optimum with the perturbed communication costs, and then applying descent to obtain a local optimum with the original communication costs.

Real-world instances

The main data used in Experiment 1 are as follows:

- Problems A, B, C, D, E and F: there are three instances with 20 tasks and 6 processors and three *instances* with 40 tasks and 12 processors; task requirements a_i range from a few up to approximately 50 units; processors capacities b_k range from 100 to 250 units; fixed costs s_k range from 1,000 to 5,000 units; communication cost matrices are very dense, with c_{ij} ranging from a few up to 50 units; and assignment costs are zero ($d_{ik} = 0$).
- Problem G: 15 tasks and 5 processors; $a_i = 1$; b_k range from 3 to 5 units; $s_k = 0$; communication cost matrices are very sparse, with c_{ij} equal to 0 or 1; and $d_{ik} = 0$.
- Problem H: 41 tasks and 4 processors; a_i range from a few up to 950 units; b_k range from 800 to 1600 units; $s_k = 0$; communication cost matrices are very sparse, with c_{ij} ranging from a few to 70 units; and $d_{ik} = 0$.

Generated data

We use the notation U[u, v] to denote an integer randomly generated from a uniform distribution defined on the interval [u, v]. The data used in Experiments 2 and 3 were generated as follows:

- Number of tasks: Experiment 2: $n = 20$, 40, 60, 80 and 100; Experiment 3: $n = 20$, 40 and 60.
- Number of processors: $m = 5$, 10 (only for $n \geq 40$), 20 (only for $n \geq 60$) and 30 (only for $n = 100$).

- $a_i \in$ U[50, 100]; let $A = \sum_{i=1}^{n} a_i$
- $b_k \in$ U[$b_{\min}$, $b_{\max}$]:
  - *loose* case: $b_{\min} = 3A/m$, $b_{\max} = 5A/m$ (on average, only a quarter of the processors may be necessary)
  - *medium* case: $b_{\min} = A/m$, $b_{\max} = 3A/m$ (on average, half of the processors may be necessary)
  - *tight* case: $b_{\min} = A/(3m)$, $b_{\max} = 7A/(3m)$ (on average, more than the available processors would be necessary, but in practice feasible solutions are often obtained)
- $s_k \in$ U[b_k , Sb_k], with S = 10, 50 and 100
- The communication cost between task i and task j is greater than 0 with a probability of 0.25. This rule gives sparse communication cost matrices, which are good for algorithm testing. Then, $c_{ij} \in$ U[50 , 100] with probability 0.25, and $c_{ij} = 0$ with probability 0.75
- $d_{ik} = 0$ (Experiment 2) and $d_{ik} \in$ U[50 , 100] (Experiment 3)

Hardware and Software

The algorithms (GVNS and HYBRID) were programmed using the C language and run on a PC Pentium IV at 2.6 GHz with 1024 Mb RAM. The computational experiments reported by Hadj-Alouane et al. (1999) were performed on an IBM RS/6000-320H (in C language), and the algorithm was run 10 times with different seeds. Ernst et al. (2006) implemented their approaches in C/C++ (using CPLEX for solving integer linear programming formulations) and ran the code on a computer with a 500MHz alpha processor.

Experimental results

The following tables (Tables 1 to 7) and figures (Figures 7 to 9) summarise the results of Experiments 1, 2 and 3. In Table 1, EJK (best lower bound and best found solution) refers to Ernst et al. (2006), and HBM (best, average and worst found solutions) refers to Hadj-Alouane et al. (1999). For each instance, the best solutions are shown in bold.

Table 1 shows that, for most of the 8 real-world instances of Hadj-Alouane et al. (1999), the GVNS algorithms outperform, in a very short solving time, the results obtained by the hybrid genetic algorithm (HBM), the column generation approaches of Ernst et al. (2006) and HYBRID. Although in

Experiment 1 HYBRID does not seem to outperform HBM, Chen and Lin (2000) carry out a wide computational experiment and show in their paper how their hybrid method generally gives better results than HBM in terms of both solution quality and computation time. Hence, if in Experiments 2 and 3 GVNS outperforms HYBRID, it could be concluded that GVNS is also better than HBM.

Insert Table 1. Results of experiment 1 (8 real-world instances)

Table 2 summarises the main results of Experiment 2 in terms of the solution quality, where we use G and H as abbreviations for GVNS and HYBRID, respectively. Table 2 shows that with a Variable Neighbourhood Search algorithm better solutions are obtained than with the hybrid method of Chen and Lin (2000). On average, our GVNS1 algorithm outperforms HYBRID for 72.2% of the instances, and in these situations the percentage of improvement is quite high (5.5% on average). Only for 27.8% of the instances are the results of the Chen and Lin (2000) algorithm better than ours, and in these cases the percentage of improvement is not very high (1.4% on average). The improvement of GVNS2 compared with GVNS1 is not very great, and it needs longer computation times (see Table 5). This leads us to conclude that the final condition of n iterations between two consecutive improvements may be too much and a shorter number of non-improvement iterations could be used instead of n.

In Table 3 and 4, corresponding results are presented according to the capacity (loose, medium or tight) and S (defining the range for fixed costs), respectively. The improvement offered by our algorithm is greater in situations in which the number of required processors (on average) is greater than the number available (loose and medium cases). This is not surprising, as these are exactly the cases in which it is possible to take greater advantage of the new neighbourhoods. In most solutions for the *tight* case, the remaining capacity of the processors may be very low, and it may be very difficult, or even impossible, to reallocate a cluster of tasks to a processor or to empty a processor, which is exactly what is attempted under neighbourhoods 3, 4 and 5. Hence, there may not be a great difference between the results of GVNS and those of HYBRID. On the other hand, the improvements offered by both algorithms are approximately the same for the different values of S (fixed costs). GVNS takes advantage of emptying a processor because this move allows it to lower fixed costs, but the HYBRID method begins with a solution in which all tasks are allocated to a cheapest processor, so the final solution is also good in terms of fixed cost.

Insert Table 2. Results of Experiment 2 (generated data set without assignment costs)

Insert Table 3. Results of Experiment 2 (generated data set without assignment costs) by capacity

Insert Table 4. Results of Experiment 2 (generated data set without assignment costs) by *S*

The final condition set for GVNS1 ensures that its computation time is always approximately equal to or shorter than the maximum HYBRID solving time. Obviously, both algorithms need more computation time when the number of tasks *n*) and the number of processors *m* grow (see Tables 5 and 6 and Figures 7 and 8), but the results confirm that the GVNS algorithm is very efficient and can be used even for large instances.

Insert Table 5. Experiment 2 (generated data set without assignment costs). Computation times as *n* varies (final condition for GVNS2 includes a maximum computation time of 50 seconds)

Insert Table 6. Experiment 2 (generated data set without assignment costs). Computation times as *m* varies (final condition for GVNS2 includes a maximum computation time of 50 seconds)

Insert Figure 7. Experiment 2 (generated data set without assignment costs). Computation times as *n* varies

Insert Figure 8. Experiment 2 (generated data set without assignment costs). Computation times as *m* varies

Finally, Tables 7 and 8 and Figure 9 summarise the main results of Experiment 3 in terms of solution quality and computation times. Again, the GVNS1 gives better solutions than the HYBRID method with comparable computation times, and GVNS2 further improves solution quality but at the expense of computation time.

Insert Table 7. Results of Experiment 3 (generated data set with assignment costs)

Insert Table 8. Computation times of Experiment 3 (generated data set with assignment costs)

Insert Figure 9. Experiment 3 (generated data set with assignment costs). Computation times as *n* varies

HYBRID may be perceived as a method that is not designed to take assignment costs into account, and thus better solutions may be expected with GVNS. Even in Experiment 2 when there are no

assignment costs, solutions are generally better for GVNS than for HYBRID, which refutes the perception that the superiority of GVNS is mainly attributed to certain of its design features that aim to reduce assignment costs. Tabu search with diversification often allows solutions to improve when allocated more computation time, and the HYBRID method could potentially benefit by allowing further iterations. Nevertheless, we do not anticipate a significant improvement in solution quality when increasing the number of iterations suggested by Chen and Lin (2000).

Conclusions

The Constrained Task Allocation Problem (CTAP), which is known to be NP-hard, consists of assigning a set of tasks to a set of processors so that the overall cost is minimised. This cost includes a fixed cost of using a processor, a task assignment cost (which may depend on the task and processor) and a communication cost between tasks that are assigned to different processors.

In this paper, a Variable Neighbourhood Search algorithm for tackling the CTAP is proposed. Three new neighbourhoods are added to the neighbourhoods traditionally used (reallocating a task and exchanging two tasks): (i) reallocating a cluster of tasks from one processor to another; (ii) reallocating a cluster of tasks from different processors to another processor; and (iii) emptying a processor by reallocating its assigned tasks to other processors. Three clustering procedures, which consider assignment and communication costs, are designed to find suitable moves within these three new neighbourhoods.

Extensive computational experiments show that the strengths of the Variable Neighbourhood Search algorithm. Specifically, it outperforms HYBRID, the previous state-of-the-art algorithm of Chen and Lin (2000) both in terms of quality solution and computation times.

Acknowledgements

The authors are grateful to James Bean and Atidel Hadj-Alouane for providing test data. The research by the second author was partly supported by INTAS, Project 03-51-5501.

References

Chen W-H., Lin, C-S. (2000). A hybrid heuristic to solve a task allocation problem. *Computers & Operations Research* **27**: 287-303.

Ernst, A., Jiang, H., Krishnamoorthy, M. (2006). Exact Solutions to Task Allocation Problems. *Management Science* **52**: 1634-1646.

Hadj-Alouane, A.B., Bean, J.C., Murty, K.G. (1999). A Hybrid Genetic/Optimization Algorithm for a Task Allocation Problem. *Journal of Scheduling* **2**: 189-201.

Hamam, Y., Hindi, K.S. (2000). Assignment of program modules to processors: A simulated annealing approach. *Eur J Opl Res* **122**: 509-513.

Hansen, P., Mladenovic, N. (2001). Variable neighborhood search: Principles and applications. European Journal of Operational Research **130**: 449-467.

Mladenovic, N., Hansen, P. (1997). Variable neighbourhood search. *Computers and Operations Reasearch* **24**: 1097-1100.

Rao, G.S., Stone, H. S., Hu, T.C. (1979). Assignment of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers* **C-28:** 291-299.

Stone,H.S.(1977). Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering* **3**: 85-93.

General Variable Neighbourghood Search (GVNS)

Generate an initial solution, x, and evaluate $f(x)$
While (no termination condition) *do*
    $u = 1$
    While ($u \leq u_{max}$) *do*
    Choose, at random, a solution $x' \in N_u(x)$
        x'' is the result of applying **VND** to x'
        If $f(x'') < f(x)$ *then*
            $x = x''$ and $u = 1$
        else
            $u=u+1$
        end if
    end while
end while
Return best found solution

Figure 1. General Variable Neighbourhood Search Algorithm, GVNS

Variable Neighbourghood Descent Algorithm (VND)

x is the initial solution for VND
While (no final condition) *do*
    $u = 1$
    While ($u \leq u_{max}$) *do*
        x' is the best solution in $N_u(x)$
        If $f(x') < f(x)$ *then*
            $x = x'$ and $u = 1$
        else
            $u=u+1$
        end if
    end while
end while
Return best found solution

Figure 2. Variable Neighbourhood Descent Algorithm, VND

Reallocate a cluster of tasks from processor k to processor l, $N_3(x)$. Determine x_3

$\alpha \in [0,1)$ is a random number

Select a task s ($s \in P_k$ and $a_s \leq b_l'$)

Initialize: $T_{kl} = \{s\}$; $b_l' = b_l' - a_s$; $b_k' = b_k' + a_s$

Set $J = \{j \mid j \in (P_k - T_{kl}), a_j \leq b_l'\}$

compute $C_j = \alpha\left(\sum_{i\in(P_l \cup T_{kl})} c_{ji} - \sum_{t\in(P_k - T_{kl})} c_{jt}\right) + (1-\alpha)(d_{jk} - d_{jl})$, for all $j \in J$

While $J \neq \varnothing$ and max $C_j > 0$, *do*

Select task $t \in J$ with C_t as large as possible

Add t to cluster: $T_{kl} = T_{kl} \cup \{t\}$, $b_l' = b_l' - a_t$ and $b_k' = b_k' + a_t$

Update: set $J = \{j \mid j \in (P_k - T_{kl}), a_j \leq b_l'\}$, and $C_j = C_j + 2\alpha c_{jt}$ for all $j \in J$

end while

x_3 is the solution resulting from the reallocation of tasks from T_{kl} to processor l

Figure 3. Procedure to find x_3

Reallocate a cluster of tasks to processor l, $N_4(x)$. Determine x_4

$\alpha \in [0,1)$ is a random number

Select a task s ($s \notin P_l$ and $a_s \leq b_l'$)

Initialize: $T_l = \{s\}$; $b_l' = b_l' - a_s$

Set $J = \{j \mid j \in J, j \notin P_l \cup T_l, a_j \leq b_l'\}$

Compute $C_j = \alpha\left(\sum_{i \in (P_l \cup T_l)} c_{ji} - \sum_{t \in (P_k - T_l)} c_{jt}\right) + (1-\alpha)\left(d_{jk} - d_{jl}\right)$, for all $j \in J$, where k is the processor to which j is currently assigned

While $J \neq \varnothing$ and max $C_j > 0$, *do*

Select task $t \in J$ with C_t as large as possible, and select k such that $t \in P_k$

Add t to cluster: $T_l = T_l \cup \{t\}$ and $b_l' = b_l' - a_t$

Update: set $J = \{j \mid j \in J, j \notin P_l \cup T_l, a_j \leq b_l'\}$ and

$$C_j = \begin{cases} C_j + 2\alpha c_{jt} & \text{for all } j \in J \cap P_k \\ C_j + \alpha c_{jt} & \text{for all } j \in J - P_k \end{cases}$$

end while

x_4 is the solution resulting from the reallocation of tasks from T_l to processor l.

Figure 4. Procedure to find x_4

Empty a processor k, $N_5(x)$. Determine x_5

$\alpha \in [0,1)$ is a random number

Initialize: $T_{kl} = \varnothing$, for all $l \neq k$

Set $J_l = \{j \mid j \in P_k, a_j \leq b_l'\}$ for all $l \neq k$

Compute $C_{jl} = \alpha\left(\sum_{i \in P_l} c_{ji} - \sum_{t \in P_k} c_{jt}\right) - (1-\alpha)\, d_{jl}$, for all $l \neq k$ and $j \in J_l$

While $J_l \neq \varnothing$, *do*

Select a task t and the corresponding processor p such that C_{tp} is as large as possible

Add t to cluster: $T_{kp} = T_{kp} \cup \{t\}$ and $b_p' = b_p' - a_t$

Update: set $J_l = \{j \mid j \in (P_k - \bigcup_{l \neq k} T_{kl}), a_j \leq b_l'\}$ for all $l \neq k$ and

$$C_{jl} = \begin{cases} C_{jl} + 2\alpha c_{jt} & \text{for all } j \in J_p \\ C_{jl} + \alpha c_{jt} & \text{for all } l \neq p \text{ and } j \in J_l \end{cases}$$

end while

x_5 is the solution resulting from the reallocation of tasks from T_{kl} to processor l, for $l \neq k$.

Figure 5. Procedure to find x_5

Initial solution, *x*

$\alpha \in [0,1)$ is a random number
Initialize: $P_k = \varnothing$, for $k=1,\ldots,m$
Sort processors by non-decreasing fixed cost (break ties at random)
Set k to be the first processor
While there are non-assigned tasks, *do*
  Set $J = \{j \mid j \in J, a_j \leq b_k'\}$
  While $J \neq \varnothing$, *do*
    Compute $C_j = \alpha\left(\sum_{i \in P_k} c_{ji}\right) - (1-\alpha) d_{jk}$, for all $j \in J$
    Select task $t \in J$ with C_t as large as possible
    Add t to processor k: $P_k = P_k \cup \{t\}$ and $b_k' = b_k' - a_t$
    Update: $J = \{j \mid j \in J, a_j \leq b_k'\}$
  while
  Select k to be the next processor
end while
Initial solution, x, is determined by $P_1,\ldots,P_m$

Figure 6. Algorithm to find initial solution

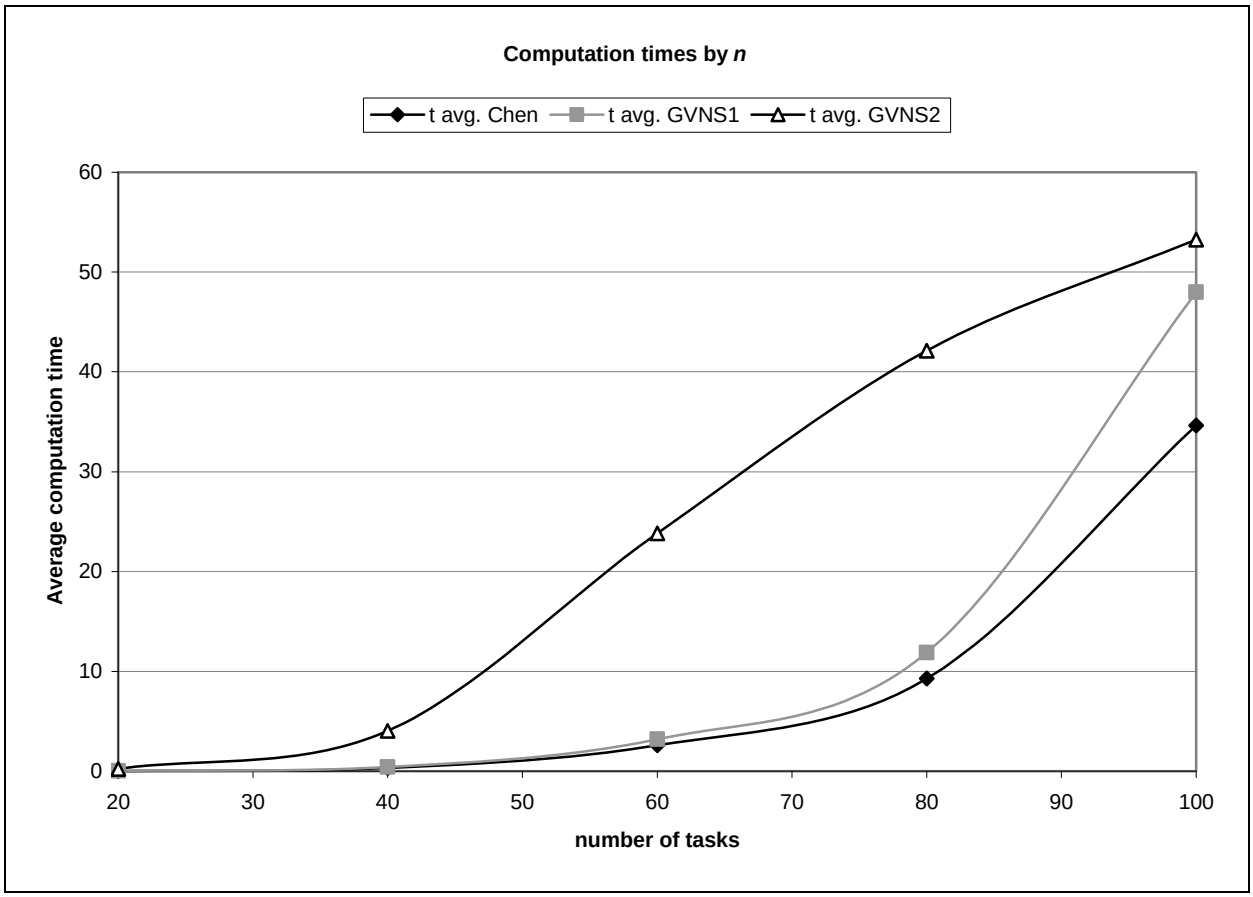

Figure 7. Experiment 2 (generated data set without assignment costs). Computation times as *n* varies

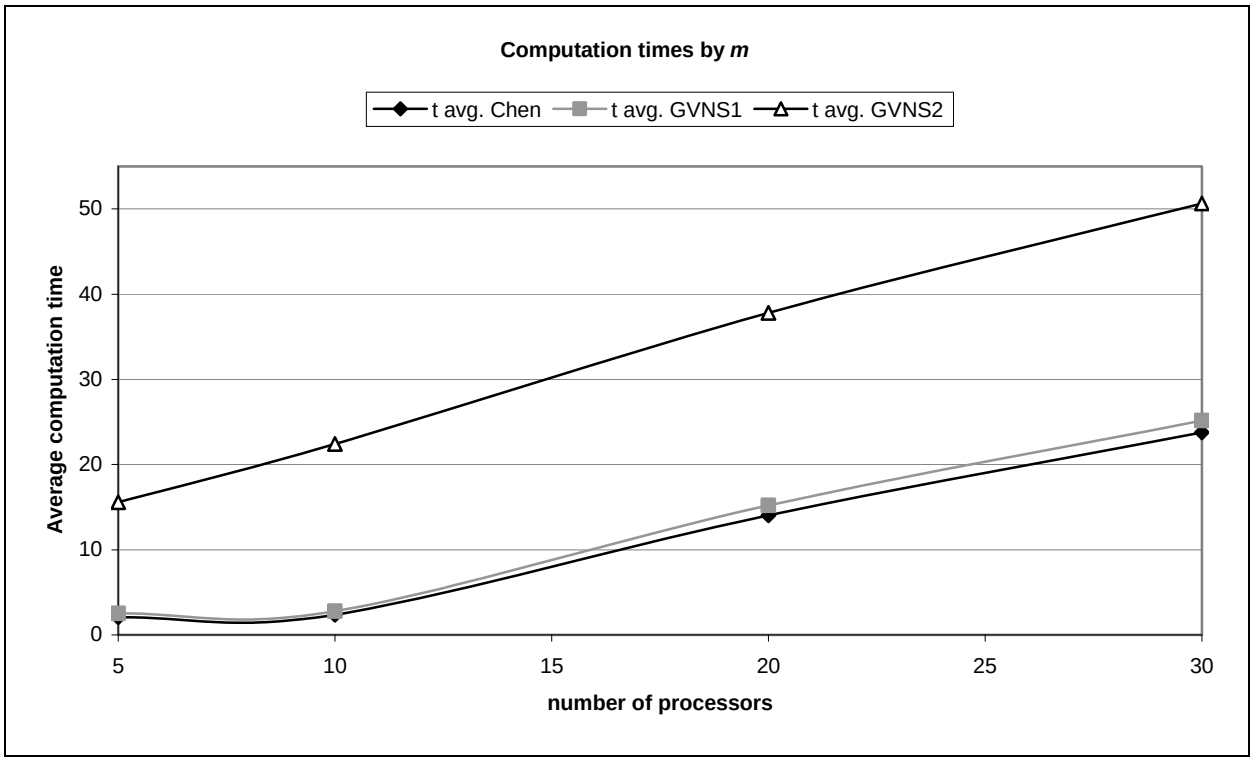

Figure 8. Experiment 2 (generated data set without assignment costs). Computation times as *m* varies

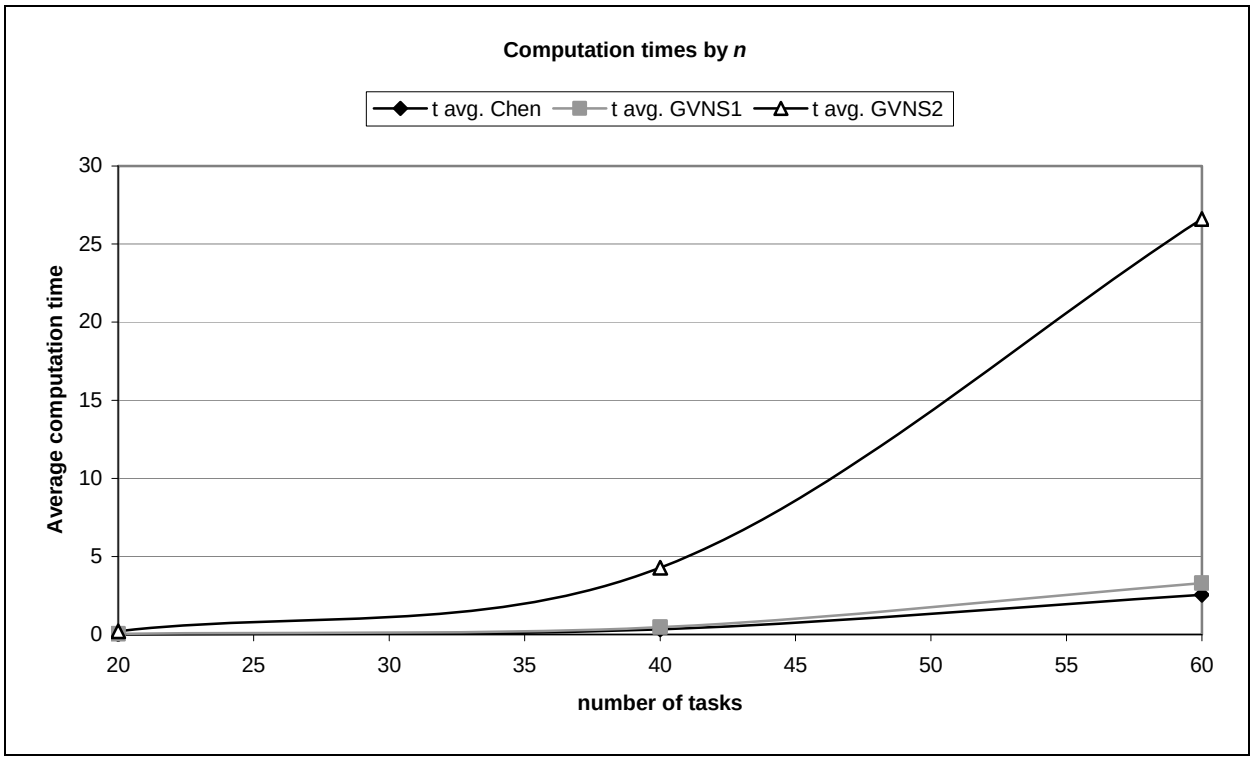

Figure 9. Experiment 3 (generated data set with assignment costs). Computation times as *n* varies

	SOLUTION (min, average, max)						TIME (min, average, max)				
Problem (*n*-*m*)	**Best Low EJK**	**Best EJK**	**HBM**	**HYBRID**	**GVNS1**	**GVNS2**	**EJK**	**HBM**	**HYBRID**	**GVNS1**	**GVNS2**
A (20-6)	13,310.37	**13,450**	13,804 13,866 13,903	13,519 15,508 15,558	**13,450** 13,940 14,263	**13,450** **13,832** 14,120	35,120	3.43 25.93 87.48	0.015 0.022 0.063	0.015 0.021 0.047	0.109 0.153 0.266
B (20-6)	11,946	**11,946**	**11,946** **11,946** 11,946	**11,946** 12,018 12,320	**11,946** 11,998 12,397	**11,946** **11,946** 11,946	671.46	10.56 28.74 73.53	0.015 0.019 0.032	0.015 0.019 0.047	0.109 0.139 0.218
C (20-6)	11,120	**11,120**	**11,120** 11,228 11,864	11,156 11,268 11,315	11,126 11,285 12,039	11,126 **11,204** 11,431	14,589.12	6.94 18.95 46.45	0.015 0.020 0.032	0.015 0.019 0.031	0.109 0.184 0.453
D (40-12)	37,662.39	39,738	39,680 39,869 41,149	41,557 41,753 41,850	39,293 39,591 40,051	**39,214** **39,385** 39,833	2,440	205.2 274.9 395.9	0.374 0.409 0.515	0.172 0.250 0.359	1.875 3.331 7.859
E (40-12)	33,438.86	38,602	36,575 37,214 38,767	37,731 38,052 38,518	35,674 36,481 38,203	**35,671** **35,901** 37,953	3,436	52.79 307.6 389.5	0.375 0.411 0.468	0.172 0.250 0.390	2.047 2.950 6.890
F (40-12)	32,126.36	35,016	35,821 36,427 36,568	36,410 36,570 36,707	**34,674** 35,575 36,360	**34,674** **34,950** 35,890	5,809.13	44.8 346.8 394.9	0.422 0.481 0.532	0.204 0.305 0.453	2.578 4.952 11.187
G (15-5)	16	**16**	**16** **16** 17	no feas	**16** 17 19	**16** **16** 17	181.1	1.31 2.73 6.87	_	0.015 0.016 0.016	0.015 0.029 0.078
H (41-4)	40	**40**	_	**40** 45 52	**40** **40** 48	**40** **40** 44	0.29	_	0.281 0.313 0.625	0.109 0.154 0.188	1.125 1.485 2.375

Table 1. Results of Experiment 1 (8 real-world instances)

Final condition GVNS	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	72.22	27.77	5.52	1.39
GVNS2	74.10	25.90	6.03	0.87

Table 2. Results of Experiment 2 (generated data set without assignment costs)

* Percentage improvement G (only if $f_G < f_H$): $100(f_H - f_G)/f_H$

** Percentage improvement H (only if $f_H < f_G$): $100(f_G - f_H)/f_G$

Final condition GVNS	capacity case	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	loose	77.77	22.22	2.96	0.77
	medium	75.00	25.00	11.70	1.74
	tight	63.88	36.11	1.39	1.52
GVNS2	loose	80.56	19.44	3.42	0.45
	medium	75.00	25.00	12.77	0.32
	tight	66.67	33.33	1.61	1.54

Table 3. Results of Experiment 2 (generated data set without assignment costs) by capacity

* Percentage improvement G (only if $f_G < f_H$): $100(f_H - f_G)/f_H$

** Percentage improvement H (only if $f_H < f_G$): $100(f_G - f_H)/f_G$

Final condition GVNS	*S* (fixed cost)	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	10	77.78	22.22	4.67	1.39
	50	66.67	33.33	6.04	1.48
	100	72.22	27.78	5.96	1.27
GVNS2	10	77.78	22.22	5.27	0.99
	50	69.44	30.56	6.49	0.87
	100	75.00	25.00	6.40	0.78

Table 4. Results of Experiment 2 (generated data set without assignment costs) by *S*

* Percentage improvement G (only if $f_G < f_H$): $100(f_H - f_G)/f_H$

** Percentage improvement H (only if $f_H < f_G$): $100(f_G - f_H)/f_G$

	Computation times (min, average, max)		
n	**HYBRID**	**GVNS1**	**GVNS2**
20	0.01 0.01 0.02	0.02 0.03 0.05	0.14 0.21 0.36
40	0.29 0.32 0.37	0.37 0.43 0.66	2.89 4.03 7.76
60	2.36 2.59 3.03	3.07 3.21 3.54	15.97 23.83 38.32
80	8.72 9.28 11.55	11.41 11.89 12.73	34.89 42.11 49.43
100	31.79 34.64 49.66	42.02 47.98 52.04	46.03 53.24 58.61

Table 5. Experiment 2 (generated data set without assignment costs). Computation times listed as *n* varies (final condition for GVNS2 includes a maximum computation time of 50 seconds)

	Computation times (min, average, max)		
m	**HYBRID**	**GVNS1**	**GVNS2**
5	1.93 2.05 2.37	2.40 2.51 2.77	12.41 15.57 22.42
10	9.96 10.56 14.06	12.06 13.26 15.20	23.59 29.02 37.81
20	15.70 16.76 23.76	21.09 23.48 25.14	34.20 42.75 50.65
30	34.52 39.45 56.50	49.14 56.61 58.98	50.00 59.20 62.10

Table 6. Experiment 2 (generated data set without assignment costs). Computation times listed as *m* varies (final condition for GVNS2 includes a maximum computation time of 50 seconds)

Final condition GVNS	% instances G better than H	% instances H better than G	% improvement G (average)*	% improvement H (average)**
GVNS1	62.96	37.04	5.99	1.03
GVNS2	62.96	37.03	7.09	0.77

Table 7. Results of Experiment 3 (generated data set with assignment costs)

* Percentage improvement G (only if $f_G < f_H$): $100(f_H - f_G)/f_H$

** Percentage improvement H (only if $f_H < f_G$): $100(f_G - f_H)/f_G$

	Computation times (min, average, max)		
n	HYBRID	GVNS1	GVNS2
20	0.01 0.01 0.03	0.03 0.04 0.06	0.15 0.20 0.33
40	0.29 0.32 0.41	0.41 0.47 0.60	3.10 4.27 8.03
60	2.32 2.53 3.10	3.14 3.29 3.77	17.14 26.60 41.37

Table 8. Computation times of Experiment 3 (generated data set with assignment costs)

FIGURES

Figure 1. General Variable Neighbourhood Search Algorithm, GVNS
Figure 2. Variable Neighbourhood Descent Algorithm, VND
Figure 3. Procedure to find x_3
Figure 4. Procedure to find x_4
Figure 5. Procedure to find x_5
Figure 6. Algorithm to find initial solution
Figure 7. Experiment 2 (generated data set without assignment costs). Computation times as *n* varies
Figure 8. Experiment 2 (generated data set without assignment costs). Computation times as *m* varies
Figure 9. Experiment 3 (generated data set with assignment costs). Computation times as *n* varies

TABLES

Table 1. Results of Experiment 1 (8 real-world instances)
Table 2. Results of Experiment 2 (generated data set without assignment costs)
Table 3. Results of Experiment 2 (generated data set without assignment costs) by capacity
Table 4. Results of Experiment 2 (generated data set without assignment costs) by *S*
Table 5. Experiment 2 (generated data set without assignment costs). Computation times as *n* varies (final condition for GVNS2 includes a maximum computation time of 50 seconds)
Table 6. Experiment 2 (generated data set without assignment costs). Computation times as *m* varies (final condition for GVNS2 includes a maximum computation time of 50 seconds)
Table 7. Results of Experiment 3 (generated data set with assignment costs)
Table 8. Computation times of Experiment 3 (generated data set with assignment costs)